CONFIDENTIAL ASCII MSX

M S X -- D O S v e r s i o n 2

The advanced disk operating system

for MSX 2 computers

Copyright (1986) ASCII Corp.

CONTENTS

		Page
1.	Introduction	3
2.	Transient Program Environment	4
2.1	Entry from MSX-DOS	4
2.2	Return to MSX-DOS	4
2.3	Page Zero Usage	5
2.4	BIOS Jump Table	8
2.5	RAM Paging	9
3.	MSX-DOS calls	11
3.1	Calling Conventions	11
3.2	Devices and Character I/O	12
3.3	File Handles	13
3.4	File Info Blocks	15
3.5	Environment Strings	18
3.6	File Control Blocks	20
4.	Screen Control Codes	23
5.	Mapper Support Routines	25
5.1	Mapper Initialization	25
5.2	Mapper Variables and Routines	25
5.3	Using Mapper Routines	26
5.4	Allocating and Freeing Segments	28
5.5	Inter-Segment Read and Write	28
5.6	Inter-Segment Calls	29
5.7	Direct paging routines	30
6.	Errors	32
6.1	Disk Errors	32
6.2	MSX-DOS Function Errors	34
6.3	Program Termination Errors	40
6.4	Command Errors	41

This document describes the interface to transient programs provided by MSX-DOS version 2.00.

Copyright (1986) ASCII Corporation
Copyright (1986) IS Systems Ltd.

1. INTRODUCTION

This manual describes the environment which MSX-DOS 2 provides for transient programs on MSX 2 computers. It is intended as a guide for writing new programs to run under MSX-DOS 2 and also to assist in converting existing CP/M and MSX-DOS 1 programs to take advantage of the advanced features.

MSX-DOS 2 provides many enhancements to the standard CP/M and MSX-DOS 1 environment, but is largely compatible with existing programs. The main features include:

MS-DOS style tree structured directories
File handles
Environment Strings
Proper error handling

Many extra DOS calls are implemented, and these are accessed via the DOS entry jump at address 5 (the 'BDOS' jump in CP/M). The descriptions of the individual functions can be found in the MSX-DOS 2 Function Specification.

Throughout this manual, the term MSX-DOS is used generally to refer to MSX-DOS 2 unless otherwise stated.

2. TRANSIENT PROGRAM ENVIRONMENT

This chapter describes the environment in which transient programs are executed under MSX-DOS, including entry and exit to the program and memory usage.

2.1 ENTRY FROM MSX-DOS

A transient program will be loaded at address 0100h, the start of the TPA (Transient Program Area), and is CALLed by MSX-DOS with the stack pointer set to a small stack within MSX-DOS. This stack is sufficient for making MSX-DOS function calls and for allowing MSX interrupts. If more stack is required (as it normally will be) then the transient program must set up its own stack in the TPA.

The contents of the Z-80 registers when a transient program is entered are undefined. The first 256 bytes of RAM will have been set up with various parameters and code as described in section 2.3.

Interrupts are enabled when a transient program is entered and should generally be left enabled. MSX-DOS function calls will generally re-enable interrupts if the transient program has disabled them.

2.2 RETURN TO MSX-DOS

A transient program can terminate itself in any of the following four ways:

1. Returning, with the original stack pointer.
2. Jump to location 0000h.
3. MSX-DOS "Program Terminate" function call 0.
4. MSX-DOS "Terminate with Error Code" function call.

The first two of these methods are identical as far as MSX-DOS is concerned, and are compatible with CP/M and MSX-DOS 1. The third method is also compatible with CP/M and MSX-DOS 1 and is equivalent to doing a "Terminate with Error Code" function call with an error code of zero.

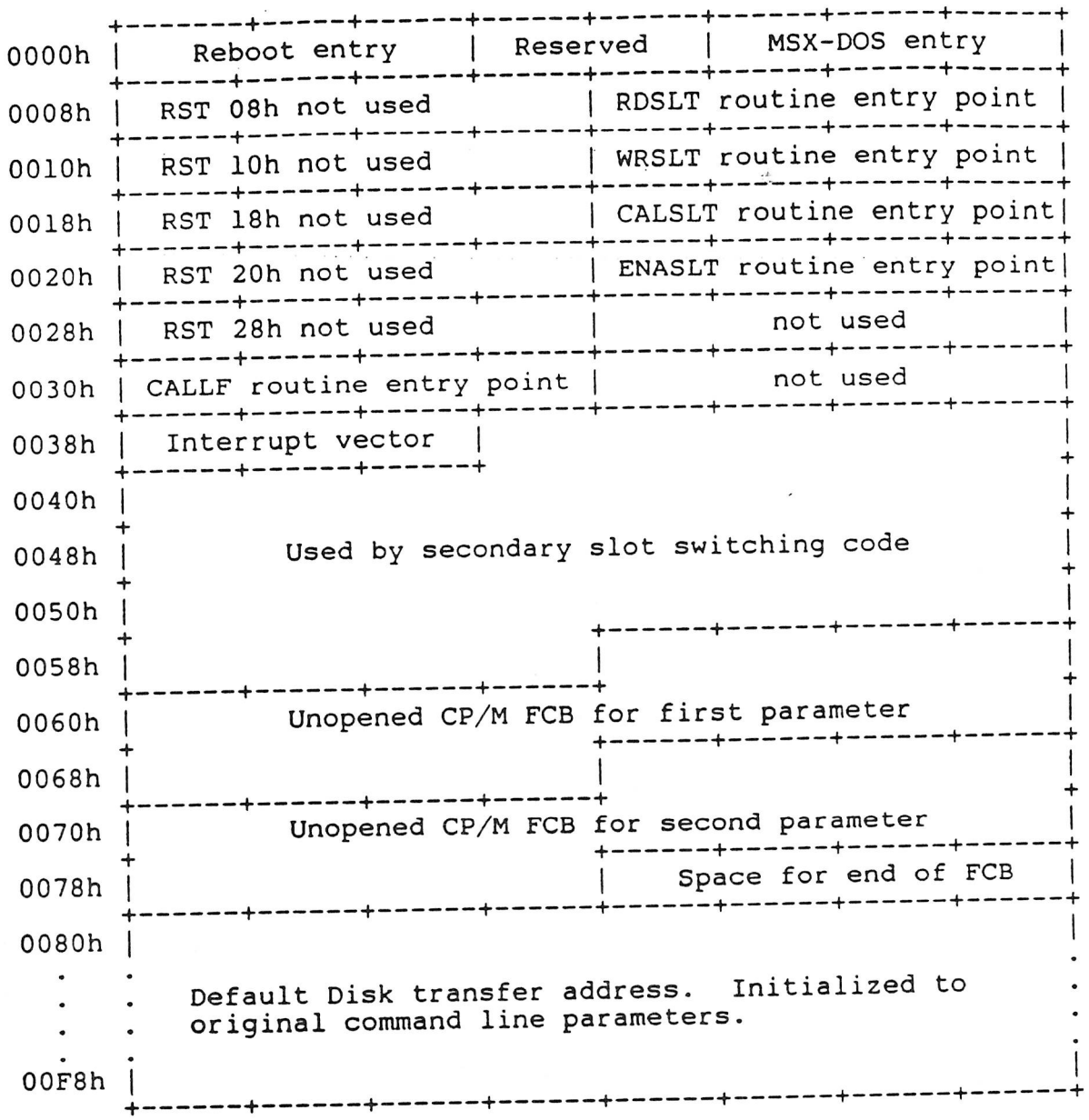
The new "Terminate with Error Code" function allows the program to return an error code to MSX-DOS, the first three terminating methods always returning an error code of zero (no error). All specially written programs and converted CP/M programs should use this new function, even for returning an error code of zero.

Various other events outside the control of a program can cause it to terminate. For example, typing "Ctrl-C" or "CTRL-STOP" at the keyboard, by the user selecting "Abort" as the response to an "Abort/Retry/Ignore" disk error message or by an error on the standard I/O channels. In these cases an an appropriate error code will be returned to MSX-DOS.

A transient program can define an "abort routine". This will be called when the program terminates by a "Program Terminate" or "Terminate with error code" function, or after an abort error (see above). How to define this routine and for what may be used is described in the MSX-DOS Function Specification.

2.3 PAGE ZERO USAGE

On entry, various parameter areas are set up for the transient program in the first 256 bytes of RAM. The layout of this area is as below and is compatible with MSX-DOS 1 and with CP/M apart from the area used for MSX slot switching calls.



At address 0000h is a jump instruction which can be used for terminating the transient program. The destination of this jump can also be used to locate the BIOS jump vector (see section 2.4). The low byte of this jump address will always be 03h for CP/M compatibility.

The two reserved bytes at addresses 0004h and 0005h are the IOBYTE and current drive/user in CP/M. Although MSX-DOS keeps the current drive byte up to date for CP/M compatibility, new programs are not recommended to use this but instead to use the "Get current drive" MSX-DOS function call. The user number and IOBYTE are not supported since I/O redirection is not done in the same way as CP/M and there is no concept of user numbers.

At address 0005h is a jump instruction to the start of the resident part of MSX-DOS which is used for making MSX-DOS calls. In addition the address of this jump defines the top of the TPA which the program may use. The size of the TPA depends on what cartridges etc. are used on the MSX machine, but is typically 54K. The low byte of the destination of this jump will always be 06h for CP/M compatibility, and the six bytes immediately preceding it will contain the CP/M version number and a serial number.

Four bytes are reserved for the user at each Z80 restart location, which is sufficient for a jump. The bytes between the restart locations however are used for the entry points to various MSX slot switching routines.

The whole area from 0038h to 005Bh is used for MSX interrupt and secondary slot switching code, and must not be modified. Note that most CP/M debuggers (such as ZSID and DDT) use address 38h as a breakpoint entry, and these programs will have to be modified to use a different restart. RST 28h is recommended.

The two FCBs set up at addresses 005Ch and 006Ch are valid unopened FCBs containing the first two command line parameters interpreted as filenames. If both filenames are to be used then the second one must be copied to a separate FCB elsewhere in memory because it will be overwritten when the first one is opened. See section 3.6 for the format of FCBs.

The whole of the command line, with the initial command removed, is stored in the default disk transfer area at address 0080h, with a length byte first and a terminating null (neither the null nor the length byte are included in the length). This string will have been upper-cased and will include any leading spaces typed to ensure CP/M compatibility.

New programs for MSX-DOS should not use the CP/M FCBs, since other MSX-DOS calls are available which are generally easier to use and which allow programs to access directories and handle path names (see section 3 for details of these facilities).

Improved methods are also available for accessing the command line. An environment string is set up which contains the command line not upper-cased. Another environment string allows programs to find out the drive, directory and filename from which they were loaded. See section 3.5 for details of these environment strings and of environment strings in general.

2.4 BIOS JUMP TABLE

The jump at address 0000h will always jump to an address whose low byte is 03h. At this address will be another jump instruction which is the second entry in a seventeen entry jump table. This corresponds exactly to the BIOS jump table in CP/M 2.2.

The first eight entries in the table are for rebooting and for character I/O. These routines are implemented with the same specification as CP/M. The remaining jumps are low level disk related functions in CP/M and have no equivalent in MSX-DOS since its filing system is totally different. These routines simply return without doing anything apart from corrupting the main registers and returning an error where possible.

MSX-DOS switches to an internal stack while executing a BIOS call and so only a small amount of space (8 bytes) is required on the user's stack.

Note that although the jump table is always on a 256 byte page boundary, it is not the "correct" distance above the top of the TPA (as defined by the contents of address 0006h) to correspond with CP/M 2.2. This should not matter to well behaved CP/M programs but it is rumoured that some programs rely on the size of the BDOS in CP/M 2.2. These programs will need modification.

The entries in the BIOS jump vector are as below:

```
xx00h - JMP  WBOOT     ;Warm boot
xx03h - JMP  WBOOT     ;Warm boot
xx06h - JMP  CONST     ;Console status
xx09h - JMP  CONIN     ;Console input
xx0Ch - JMP  CONOUT    ;Console output
xx0Fh - JMP  LIST      ;List output
xx12h - JMP  PUNCH     ;Punch (auxiliary) output
xx15h - JMP  READER    ;Reader (auxiliary) input
```

```
xx18h - JMP  RETURN    ;Home in CP/M
xx1Bh - JMP  RETURN    ;Select disk in CP/M
xx1Eh - JMP  RETURN    ;Set track in CP/M
xx21h - JMP  RETURN    ;Set sector in CP/M
xx24h - JMP  RETURN    ;Set DMA address in CP/M
xx27h - JMP  RETURN    ;Read sector in CP/M
xx2Ah - JMP  RETURN    ;Write sector in CP/M
xx2Dh - JMP  LSTST     ;List status
xx30h - JMP  RETURN    ;Sector translate in CP/M
```

2.5 RAM PAGING

When a transient program is loaded, the mapper RAM slot will be enabled in all four pages and the four RAM segments which make up the basic 64k will be paged in. There will be MSX BIOS ROM compatible slot handling entry points available in page-0 and various mapper support routines available in page-3 (see section 5 for specifications of these).

A program may do any slot switching and paging which it likes while it is running and need not restore either the slot selections or the RAM paging before it exits, since COMMAND2.COM will handle this. A program must of course take the usual precautions with the interrupt entry point if it alters page-0, and must never alter page-3 (nothing is allowed to do that!).

Although a transient program is free to do slot switching, it must always ensure that whenever it does an MSX-DOS function call or an MSX-DOS BIOS call, the mapper RAM slot is selected in pages 0, 2 and 3. This is because the function entry code does RAM paging but does not do any slot switching as this would slow down function entry too much. Page-1 can contain any slot when doing a function call and will be preserved. Any parameters can be passed in page-1, except that environment strings and disk transfer areas must be in the mapper RAM slot.

Any RAM segments can be selected in pages 0, 1 and 2 when an MSX-DOS function call or an MSX-DOS BIOS function call is made, and also the stack can be in any page. The current paging state will be preserved by all function calls even in error conditions. Any disk transfers will be done to the RAM segments which are paged in when the function call is made, even if they are not the original TPA segments.

If a transient program wants to use more RAM than the TPA then it can use the mapper support routines (described in section 5) to obtain more RAM. Before using any RAM other than the four TPA segments, the program must ask the mapper routines to allocate a new segment. This ensures that there is no contention with the program trying to use a segment which is already in use (by the RAM disk for example). The segments should normally be allocated as "user segments" since these will automatically be freed when the program terminates. "system segments" should only be allocated if it is necessary for them to remain in use after the transient program has terminated.

Having allocated additional segments, the program may page them in and use any of the mapper support routines to access them. It will normally be necessary for a transient program to remember the segment numbers of the TPA segments in order to page them back in when they are required. The segment numbers will normally be 0, 1, 2 and 3 but this must NOT be assumed by transient programs, they must use the "GET_Pn" mapper routines to find out the segment numbers before paging anything else in.

3. MSX-DOS FUNCTION CALLS

3.1 CALLING CONVENTIONS

MSX-DOS function calls are made by putting the function code in register C with any other parameters required in the other main registers and then executing a "CALL 5" instruction. The operation will be performed and the results will be returned in various registers depending on the function.

Generally all the main registers (AF, BC, DE and HL) will be corrupted by MSX-DOS calls or will return results. The alternate register set (AF', BC', DE' and HL') are always preserved, whilst the index registers (IX and IY) are preserved except in the one or two cases where they return results.

Only a small amount of space (8 bytes) is needed on the transient program's stack because MSX-DOS switches to an internal stack when it is called. The mapper RAM slot must be enabled in pages 0, 2 and 3 when a function call is made.

For compatibility all functions which have a CP/M counterpart return with A=L and B=H. Frequently A returns an error flag, with zero indicating success and 01h or FFh indicating failure.

All of the new MSX-DOS functions (function code above 40h) return with an error code in A and any other results in the other main registers. An error code of 0 means no error, whilst any non-zero code means an error occurred, the exact nature of which can be found by looking at the value. A list of error codes and messages is given in section 6. An "explain" function is provided which will give an ASCII explanation string for an error code.

The actual functions available are documented in the MSX-DOS Function Specification.

3.2 DEVICES AND CHARACTER I/O

Wherever a filename is given to an MSX-DOS function, a device name may also be given. These devices are used for character based I/O and allow a program to access a disk file or a character device in exactly the same way without having to know which it is using.

The syntax of device names is identical to that of filenames so programs do not need any special handling to use device names. This applies both to the new MSX-DOS 2 functions and the CP/M FCB functions. The reserved filenames used for devices are:

CON - screen output, keyboard input
PRN - printer output
LST - printer output
AUX - auxiliary output/input
NUL - null device

When any of these appear as the main part of a filename, it actually refers to the device; the extension is ignored. Most function calls that use files can also use devices. For example, a filename of CON can be successfully given to the "rename file" function or the "delete file" function. No error will be returned but the device will be unaffected.

The AUX device mentioned above does not do anything by default, but a routine may be hooked into it so that it refers for example to a serial driver. The NUL device does not actually do anything; output characters are ignored and an end-of-file is always input. The LST and PRN devices are identical.

The CON device is used to read from the keyboard or write to the screen. When reading from the CON device, the system will read a line at a time allowing the user to use line editing facilities. Only when the user types a CR (carriage return) will the line be entered. End of input is signified by a Ctrl-Z character at the start of a line.

The system automatically opens several file handles to these standard devices (see section 3.3 for details). These file handles may be used by programs for accessing the standard devices. Alternatively a program can do character I/O by using the traditional CP/M character functions (functions 01h...0Bh). These two methods are both acceptable but they should not normally be mixed since they use separate buffering schemes and so characters can get lost in these buffers.

For future versions of the system which may support command line redirection, both of these methods (standard file handles and character functions) will be redirected. However it is preferable to use the standard file handles and to read and write in large blocks because when accessing disk files these will be very much faster than using the character functions.

For compatibility with future versions which allow command line redirection, programs may sometimes need to do screen output and keyboard input which will bypass any redirection. For example disk error handling routines may need to do this. To facilitate this, there is a function provided which allows redirection of the character functions to be temporarily cancelled. This is described in the "Function Specification" document (function number 70h).

3.3 FILE HANDLES

File handles are the method by which files are read from and written to using the new MSX-DOS functions. File handles may also be used to manipulate files in other ways.

A file handle is an 8 bit number which refers to a particular open file or device. A new file handle is allocated when the "open file" or "create file" function is used. The file handle can be used to read and write data to the file and remains in existence until the "close file handle" or "delete file handle" function is called. Other operations can be done on files using file handles, such as changing the attributes or renaming the files to which they refer.

Whenever MSX-DOS allocates a new file handle, it will use the lowest number available. The maximum file handle number in the current version is 63. In future versions this may be increased but will never be greater than 127, so file handles can never be negative.

Space for the internal data structures used for file handles is allocated dynamically within a 16K RAM segment (the "data segment") so there is no fixed limit to the number of file handles which can be open at one time. This segment is kept outside the TPA, so anything stored there does not reduce TPA size. As well as keeping internal file handle information, the system also keeps disk buffers and environment strings in the data segment.

Various file handles are pre-defined and are already open when a transient program is executed. These file handles refer to the standard input and output devices (see section 3.2). The "traditional" CP/M style MSX-DOS character I/O functions actually refer to these file handles.

A transient program actually gets a copy of the standard input and output file handles which the command interpreter was using, rather than the originals. This means that the program can freely close these file handles and re-open them to different destinations and need not reset them before terminating.

The default file handles and their destinations are:

0 - Standard input (CON)
1 - Standard output (CON)
2 - Standard error input/output (CON)
3 - Standard auxiliary input/output (AUX)
4 - Standard printer output (PRN)

When the command interpreter is about to execute a command (for example a transient program), it executes a "fork" function call. This informs the system that a new program is being executed as a "subroutine" and amongst other things, all of the currently open file handles are duplicated, so that the new program will be using copies of the original handles, rather than the command interpreter's.

If the transient program changes any of the file handles, by closing any existing ones or opening new ones, it will be the program's own set of file handles which are modified, the original set will remain unaltered. After the program has terminated, the command interpreter executes a "join" function call, passing to it a process id which was returned from the original "fork". This tells the system that the new program has terminated and so all its file handles can be thrown away.

Reference counts are kept of how many copies of each handle there are which enables the system to tidy up any file handles which are no longer needed when a program terminates. This ensures that the system will not run out of file handles because of badly behaved programs not closing them.

These "fork" and "join" functions are available for user programs if they find them useful. In addition to tidying up file handles, "join" will also free up any user allocated RAM segments which the program has not freed.

3.4 FILE INFO BLOCKS

All new MSX-DOS functions that act on files on disk can be passed a simple pointer to a null-terminated string (called an ASCIIZ string), which can contain a drive, path and unambiguous filename. These are typically the operations which a transient program will perform, often through a high level language interface. The Command Specification gives details of these.

To any of these ASCIIZ functions, a File Info Block (FIB) may passed instead. FIBs are used for more complex operations such as the searching of directories for unknown files or sub-directories. These operations are typically only used by the command interpreter and utility commands and only sophisticated transient programs are likely to use them.

A FIB is a 64 byte area of the user's memory which contains information about the directory entry on disk for a particular file or sub-directory. The information in a FIB is filled in by the new MSX-DOS "find" functions ("Find First", "Find new" and "Find Next"). The format of a File Info Block is as follows:

```
    0 - Always 0FFh
 1..13 - Filename as an ASCIIZ string
   14 - File attributes byte
15..16 - Time of last modification
17..18 - Date of last modification
19..20 - Start cluster
21..24 - File size
   25 - Logical drive
26..63 - Internal information, must not be modified
```

The 0FFh at the start of the fileinfo block must be there to distinguish it from a pathname string, since some functions can take either type of parameter.

The filename is stored in a suitable format for directly printing, and is in the form of an ASCIIZ string. Any spaces will have been removed, the filename extension (if present) will be preceded by a dot and the name will have been uppercased. If the entry is a volume label then the name will be stored without the "." separator, with spaces left in and not uppercased.

The file attributes byte is a byte of flags concerning the file. The format of this byte is:

Bit 0 - READ ONLY. If set then the file cannot be written to or deleted, but can be read, renamed or moved. Does not apply to sub-directories.

Bit 1 - HIDDEN FILE. If set then the file will only be found by the "Find First" function if the "hidden file" bit is set in the search attributes byte. All the commands implemented by the command interpreter that access files and directories on disk can take a "/H" option which allows the command to find hidden files.

Bit 2 - SYSTEM FILE. As far as MSX-DOS functions are concerned, this bit has exactly the same effect as the "HIDDEN FILE" bit except that the "Find New" and "Create" function calls will not automatically delete a system file. None of the commands implemented by the command interpreter allow system files to be accessed.

Bit 3 - VOLUME NAME. If set then this entry defines the name of the volume. Can only occur in the root directory, and only once. All other bits are ignored.

Bit 4 - DIRECTORY. If set then the entry is a sub-directory rather than a file and so cannot be opened for reading and writing. Only the hidden bit has any meaning for sub-directories.

Bit 5 - ARCHIVE BIT. Whenever a file has been written to and closed this bit is set. This bit can be examined by, for example, the XCOPY command to determine whether the file has been changed.

Bit 6 - Reserved (always 0).

Bit 7 - DEVICE BIT. This is set to indicate that the FIB refers to a character device (eg. "CON") rather than a disk file. All of the other attributes bits are ignored.

The time of last modification is encoded into two bytes as follows:

Bits 15..11 - HOURS (0..23)
Bits 10...5 - MINUTES (0..59)
Bits 4...0 - SECONDS/2 (0..29)

The date of last modification is encoded into two bytes as follows. If all bits are zero then there is no valid date set.

Bits 15...9 - YEAR (0..119 corresponding to 1980..2099)
Bits 8...5 - MONTH (1..12 corresponding to Jan..Dec)
Bits 4...0 - DAY (1..31)

The file size is a 32 bit number stored with the lowest byte first, and is zero for sub-directories.

The logical drive is a single byte drive number, with 1 corresponding to A:, 2 to B: etc. It will never be zero, since if zero was specified in the original function, this means the default drive and the driven number of the default drive will be filled in here.

The internal information tells MSX-DOS where on the disk the directory entry is stored. This enables functions to which the fileinfo block is passed to operate on the directory entry, for example deleting it, renaming it or opening it. Data stored here also enables the "find next" function to carry on the search to find the next matching file. The user should not access or modify the internal information at all.

Fileinfo blocks are filled in by the "find first", "find new" and "find next" MSX-DOS functions. Each of these functions locates a directory entry and fills in the fileinfo block with the relevant information.

In the case of "find first" a directory will be searched for the first entry which matches a given filename and which has suitable attributes (see the Function Specification for details). "Find next" carries on a search started by a previous "find first" function and updates the fileinfo block with the next matching entry.

"Find new" is just like "find first" except that instead of looking for a matching entry, it will create a new one and then return a fileinfo block just as if "find first" had found it.

Having created a fileinfo block using one of the "find" functions there are two ways in which it can be used. The first way is to simply use the information which it contains such as the filename and size. For example the "DIR" command simply prints out the information on the screen.

The more interesting way of using a fileinfo block is to pass it back to another MSX-DOS function in order to carry out some operation on the directory entry. Many of the MSX-DOS functions described in the Function Specification take a pointer in register DE which can either point to a drive/path/file string or a fileinfo block. In either case a particular file or directory is being specified for the function to act on.

The functions which can take such a parameter are "Delete File or Subdirectory", "Rename file or Subdirectory", "Move File or Subdirectory", "Get/Set File Attributes", "Get/Set File Date and Time" and "Open File". All of these carry out the obvious function on the specified file or directory.

A fileinfo block can also be passed to a "find first" or "find new" function in place of the drive/path/file string. In this case the fileinfo block must refer to a directory rather than a file and a filename string must also be passed in register HL (typically null which is equivalent to "*.*"). The directory specified by the fileinfo block will be searched for matches with the filename, subject to the usual attribute checking. This feature is necessary for the command interpreter so that a command such as "DIR A:UTIL" can have the required action if UTIL is a directory.

3.5 ENVIRONMENT STRINGS

MSX-DOS maintains a list of "environment strings" in it's data segment. An environment string is a named item which has a value associated with it. Both the name and the value are user-defined. Environment strings are accessed at the function call level via the "Get Environment String", "Set Environment String" and "Find Environment String" functions.

The name of an environment string is a non-null string consisting of any characters that are valid in filenames. The name can be up to 255 characters long. The characters of the name are upper-cased when the string is defined, although for name comparisons case is not significant.

The value of an environment string consists of a string of non-null characters and can be up to 255 characters long. If the value of an environment string is set to a null string, then the name is removed from the list of environment strings. Similarly, if the value of an environment string that has not been defined is read, then a null string is returned. The value is not upper-cased and no interpretation is placed on the characters in the value string.

When a transient program is loaded and executed from COMMAND.COM, two special environment strings are set up, which it can read.

An environment string called PARAMETERS is set up to the command line not including the actual command name. This is similar to the one set up at 80h for CP/M compatibility, but has not been upper-cased.

Another environment string called PROGRAM is also set up and this is the whole path used to locate the program on disk. The drive is first, followed by the path from the root and then the actual filename of the program. The drive, path and filename can be separated if desired using the "Parse Pathname" function call.

The PROGRAM environment string has several uses. The main use is that a program can use it to load overlay files from the same directory as it was loaded from. The last item in PROGRAM (ie. the actual program filename) is replaced with the name of the overlay file, and then the new string can be passed to any of the new MSX-DOS functions that take ASCIIZ strings (such as "Open File").

Note that some CP/M programs are capable of loading and running transient programs, and in this case they obviously will not have set up the PROGRAM and PARAMETERS environment strings, and they will in fact still be set up from when the CP/M program was loaded. If a program wishes to use PROGRAM and PARAMETERS and still be loadable from a CP/M program, then it can look at a variable called "LOAD_FLAG", which is in page 3 at address 0F331h. It is set to zero on every MSX-DOS function call but is set to non-zero immediately before a transient program is executed by the command interpreter. Similarly, if a new transient program has the ability to load other transient programs and it sets up PROGRAM and PARAMETERS, then it should also set LOAD_FLAG to non-zero.

Another special environment string is one called APPEND. This can be set up by the user from the command interpreter and is used by the CP/M "Open File (FCB)" function. When this function call is done and the file is not found, an alternative directory, specified by APPEND, is searched. It is not anticipated however that new transient programs will use this function call or the APPEND environment string.

Several environment strings are set up by the command interpreter when it starts up and are altered by the user to control various system features and options, and it may be useful for transient programs to read some of these. For example, it may be useful to read the PATH environment string or the DATE and TIME environment strings if the program prints out dates and times. The Command Specification contains details of these default environment strings.

3.6 FILE CONTROL BLOCKS

It is not anticipated that specially written MSX-DOS 2 transient programs or modified MSX-DOS 1 or CP/M programs will use the CP/M-compatible FCB functions, but the format of the FCBs used for these functions is given here for reference. This format is, of course, very similar to the FCBs used by CP/M and MSX-DOS 1 but the use of some of the fields within the FCB are different (though generally compatible).

A basic FCB is 33 bytes long. This type of FCB can be used for file management operations (delete, rename etc.) and also for sequential reading and writing. The random read and write functions use an extra 3 bytes on the end of the FCB to store a random record number. The MSX-DOS 1 compatible block read and write functions also use this additional three (or in some cases four) bytes - see the Function Specification for details.

The layout of an FCB is given below. A general description of each of the fields is included here. The individual function description given in the Function Specification details of how the fields are used for each function where this is not obvious.

00h Drive number 1...26. 0 => default drive. Must be set up in all FCBs used, never modified by MSX-DOS function calls (except "Open File" if APPEND was used).

01h...08h Filename, left justified with trailing blanks. Can contain "?" characters if ambiguous filename is allowed (see Function Specification). When doing comparisons case will be ignored. When creating new files, name will be uppercased.

09h...0Bh Filename extension. Identical to filename. Note that bit-7 of the filename extension characters are NOT interpreted as flags as they are in CP/M.

0Ch Extent number (low byte). Must be set (usually to zero) by the transient program before open or create. It is used and updated by sequential read and write, and also set by random read and write. This is compatible with CP/M and MSX-DOS 1.

0Dh Not used. Zeroed by "open" and "create".

0Eh Extent number (high byte) for CP/M functions. Zeroed by open and create. For sequential read and write it is used and updated as an extension to the extent number to allow larger files to be accessed. Although this is different from CP/M it does not interfere with CP/Ms use of FCBs and is the same as MSX-DOS 1.

Record size (low byte) for MSX-DOS 1 compatible block functions. Must be set to the required record size before using the block read or write functions.

0Fh Record count for CP/M functions. Set up by open and create and modified when necessary by sequential and random reads and writes. This is the same as CP/M and MSX-DOS 1.

Record size (high byte) for MSX-DOS 1 compatible block functions. Must be set to the required record size before using the block read and write functions.

10h...13h File size in bytes, lowest byte first. File size is exact, not rounded up to 128 bytes. This field is set up by open and create and updated when the file is extended by write operations. Should not be modified by the transient program as it is written back to disk by a close function call. This is the same as MSX-DOS 1 but different from CP/M which stores allocation information here.

14h...17h Volume-id. This is a four byte number identifying the particular disk which this FCB is accessing. It is set up by open and create and is checked on read, write and close calls. Should not be modified by the program. Note that this is different from MSX-DOS 1 which stores the date and time of last update here, and from CP/M which stores allocation information.

18h...1Fh Internal information. These bytes contain information to enable the file to be located on the disk. Should not be modified at all by the transient program. The internal information kept here is similar but not identical to that kept by MSX-DOS 1 and totally different from CP/M.

20h Current record within extent (0...127). Must be set (normally to zero) by the transient program before first sequential read or write. Used and modified by sequential read and write. Also set up by random read and write. This is compatible with CP/M and MSX-DOS 1.

21h...24h Random record number, low byte first. This field is optional, it is only required if random or block reads or writes are used. It must be set up before doing these operations and is updated by block read and write but not by random read or write. Also set up by the "set random record" function.

For the block operations, which are in MSX-DOS 1 but not in CP/M, all four bytes are used if the record size is less than 64 bytes, and only the first three bytes are used if the record size is 64 bytes or more. For random read and write only the first three bytes are used (implied record size is 128 bytes). This is compatible with CP/M and with MSX-DOS 1.

4. SCREEN CONTROL CODES

Below is a list of all control codes and escape sequences which may be used when doing character output by MSX-DOS character functions, BIOS calls or writing to the device CON. These are all compatible with MSX-DOS 1 and contain the VT-52 control codes.

The screen is 24 lines of 2 to 80 characters. When a printing character is displayed the cursor is moved to the next position and to the start of the next line at the end of a line. If a character is written in the bottom right position then the screen will be scrolled to allow the cursor to be positioned at the start of the next line. The letters in escape sequences must be in the correct case, the spaces are inserted for readability they are not part of the sequence. Numbers (indicated by <n> or <m>) are included in the sequence as a single byte usually with an offset of 20h added.

Although the cursor on and cursor off escape sequences are listed here, they are not generally useful to transient programs. This is because the MSX-DOS BIOS contains buffering code to reduce the amount of slot switching involved is writing to the screen. This speeds up screen accessing considerably but means that the cursor is permanently on, and is switched off when the BIOS flushes its buffer to the screen.

Ctrl-G 07h = Bell.

Ctrl-H 08h = Cursor left, wraps around to previous line, stop at top left of screen.

Ctrl-I 09h = Tab, overwrites with spaces up to next 8th column, wraps around to start of next line, scrolls at bottom right of screen.

Ctrl-J 0Ah = Line feed, scrolls if at bottom of screen.

Ctrl-K 0Bh = Cursor home.

Ctrl-L 0Ch = Clear screen and home cursor.

Ctrl-M 0Dh = Carriage return.

Ctrl-[1Bh = ESC - see below for escape sequences.

Ctrl-\ 1Ch = Cursor right, wrap around to next line, stop at bottom right of screen.

Ctrl-] 1Dh = Cursor left, wrap around to previous line, stop at top left of screen.

Ctrl-^ 1Eh = Cursor up, stop at top of screen.

Ctrl-_ 1Fh = Cursor down, stop at bottom of screen.

7Fh = Delete character and move cursor left, wrap around to previous line, stop at top of screen.

Esc A = Cursor up, stops at top of screen.

Esc B = Cursor down, stops at bottom of screen.

Esc C = Cursor right, stops at end of line.

Esc D = Cursor left, stops at start of line.

Esc E = Clear screen and home cursor.

Esc H = Cursor home.

Esc J = Erase to end of screen, don't move cursor.

Esc j = Clear screen and home cursor.

Esc K = Erase to end of line, don't move cursor.

Esc L = Insert a line above cursor line, scroll rest of screen down. Leave cursor at start of new blank line.

Esc l = Erase entire line, don't move cursor.

Esc M = Delete cursor line, scrolling rest of screen up. Leave cursor at start of next line.

Esc x 4 = Select block cursor.

Esc x 5 = Cursor off.

Esc Y <n><m>= Position cursor at row <n> column <m>. Top left of screen is n=m=20h (ASCII space).

Esc y 4 = Select underscore cursor.

Esc y 5 = Cursor on.

5. MAPPER SUPPORT ROUTINES

The disk ROM contains routines to provide support for the memory mapper chip. This allows MSX application programs or MSX-DOS transient programs to utilize more than the basic 64k of memory, without conflicting with the RAM disk or any other system software. It also allows programs to isolate themselves from the mapper chip hardware, and thus remain compatible with future machines which may use different hardware such as the Z800 processor.

5.1 MAPPER INITIALIZATION

When the DOS kernel is initialized it checks that there is a mapper chip available, and that there is at least 128k of RAM available. The MSX page-3 RAM containing all the system variables must be in the mapper RAM slot. If there is no suitable mapper chip then the system displays an error message on the screen and halts.

Next it builds up a table of all the available 16k RAM segments except for the four which make up the basic 64k. The two highest numbered segments are marked as allocated to the system, one for the DOS kernel code and one for the DOS kernel workspace. All other segments (at least two) are marked as free initially.

5.2 MAPPER VARIABLES AND ROUTINES

The mapper support routines use some variables in the DOS fixed work area. These are in an area which was reserved in version 1 and were always zero. The addresses are as follows:

0F313h	DOS_VER	DOS version number. 00h => prior to version 2, 20h => version 2.00, future versions will have higher version numbers.
0F314h	MAX_SEG	Total number of 16k RAM segments in addition to the basic 64k. Will always be in the range 4...252.
0F315h	FRE_SEG	Number of free 16k RAM segments.
0F316h	SYS_SEG	Number of 16k RAM segments allocated to the system. Will always be at least 2.

0F317h	USR_SEG	Number of 16k RAM segments allocated to the user program. Initially zero.

These variables should only ever be read by user programs, they should never be modified. They are kept updated by the mapper support routines.

A program uses the mapper support code by calling various subroutines. These are accessed through a vector table which is located at the bottom of the DOS fixed work area at 0F1C9h. The addresses of these routines are as follows:

0F1C9h	ALL_SEG	Allocate a 16k segment.
0F1CCh	FRE_SEG	Free a 16k segment.
0F1CFh	RD_SEG	Read byte from address A:HL to A.
0F1D2h	WR_SEG	Write byte from E to address A:HL.
0F1D5h	CAL_SEG	Inter-segment call. Address in IYh:IX
0F1D8h	CALLS	Inter-segment call. Address in line after the call instruction.
0F1DBh	PUT_PH	Put segment into page (HL).
0F1DEh	GET_PH	Get current segment for page (HL)
0F1E1h	PUT_P0	Put segment into page 0.
0F1E4h	GET_P0	Get current segment for page 0.
0F1E7h	PUT_P1	Put segment into page 1.
0F1EAh	GET_P1	Get current segment for page 1.
0F1EDh	PUT_P2	Put segment into page 2.
0F1F0h	GET_P2	Get current segment for page 2.
0F1F3h	PUT_P3	Not supported since page-3 must never be changed. Acts like a "NOP" if called.
0F1F6h	GET_P3	Get current segment for page 3.

5.3 USING MAPPER ROUTINES

Before a program attempts to use the mapper routines, it should check that it is running on a machine with the mapper support routines. This can be done by first testing address 0FFA7h. If this is set to a Z-80 "RET" instruction (0C9h) then there is no disk system and therefore no mapper support. If this is not a "RET" then the "MAX_SEG" variable should be checked. If this is zero then the disk system is version 1 and there is no mapper support.

A program can request a 16k RAM segment at any time by calling the "ALL_SEG" routine. This either returns an error if there are no free segments, or the segment number of a new segment which the program can use. A program must not use any segment which it has not explicitly allocated, except for the four segments which make up the basic 64k of RAM.

A segment can be allocated either as a user segment or as a system segment. User segments will be automatically freed when the program terminates, whereas system segments are never freed unless the program frees them explicitly. Normally, programs should allocate user segments.

RAM segments can be accessed by the "RD_SEG" and "WR_SEG" routines which read and write bytes to specified segments. The routines "CAL_SEG" and "CALLS" allow inter-segment calls to be done in much the same way as inter-slot calls in the current MSX system, but they are very much faster.

Routines are provided to explicitly page a segment in, or to find out which segment is in a particular page. There are routines in which the page (0...3) is specified by the top two bits of an address in HL ("PUT_PH" and "GET_PH"). And there are also specific routines for accessing each page ("GET_Pn" and "PUT_Pn"). These routines are very fast so a program should not suffer in performance by using them.

Note that page-3 should never be altered since this contains the mapper support routines and all the other system variables. Also great care must be taken if page-0 is altered since this contains the interrupt entry point. Pages 1 and 2 can be altered in any way.

None of the mapper support routines will disturb the slot selection mechanism at all. For example when "PUT_P1" is called, the specified RAM segment will only appear at address 4000h...7FFFh if the mapper slot is selected in page-1. The mapper slot can be determined by examining the variable "RAMAD3" at address 0F344h in the DOS work area. The "RD_SEG" and "WR_SEG" routines will always access the RAM segment regardless of the current slot selection in the specified page, but the mapper RAM slot must be selected in page-2.

5.4 ALLOCATING and FREEING SEGMENTS

The following two routines can be called to allocate or free segments. All registers apart from AF are preserved. An error is indicated by the carry flag being set on return. The slot selection and RAM paging may be in any state when these routines are called and both will be preserved. The stack must not be in page-1 or page-2 when either of these routines are called.

A program must not use any segment (apart from the four which make up the basic 64k) unless it has specifically allocated it, and must not continue to use a segment after it has been freed.

A segment may be allocated either as a user or a system segment. The only difference is that user segments will be automatically freed when the program terminates whereas system segments will not be. In general a program should allocate a user segment unless it needs the data in the segment to outlast the program itself. User segments are always allocated from the lowest numbered free segment and system segments from the highest numbered one.

An error from "allocate segment" usually indicates that there are no free segments, although it can also mean that an invalid parameter was passed in register A. An error from "free segment" indicates that the specified segment number does not exist or is already free.

```
ALL_SEG - Parameters:   A=0 => allocate user segment
                        A=1 => allocate system segment
                        B=0 for future compatibility
          Results:      Carry set => no free segments
                        Carry clear => segment allocated
                                       A=new segment number

FRE_SEG - Parameters:   A=segment number to free
                        B=0 for future compatibility
          Returns:      Carry set => error
                        Carry clear => segment freed OK
```

5.5 INTER-SEGMENT READ AND WRITE

The following two routines can be called to read or write a single byte from any mapped segment. The calling sequence is very similar to the inter-slot read and write routines provided by the MSX system ROM. All registers apart from AF are preserved and no checking is done to ensure that the segment number is valid.

The top two bits of the address are ignored, since the segment number specifies a 16k segment which could appear in any of the four pages. The data will be read or written from the correct segment regardless of the current paging or slot selection in that page, but note that the mapper RAM slot must be selected in page-2 when either of these routines are called. This is so that the routines do not have to do any slot switching and so can be fast. Also the stack must not be in page-2.

```
RD_SEG -  Parameters:   A = segment number to read from
                       HL = address within this segment
          Results:      A = value of byte at that address
                        All other registers preserved

WR_SEG -  Parameters:   A = segment number to write to
                       HL = address within this segment
                        E = value to write
          Returns:      A = corrupted
                        All other registers preserved
```

5.6 INTER-SEGMENT CALLS

Two routines are provided for doing inter-segment calls. These are modelled very closely on the two inter-slot call routines provided by the MSX system ROM, and the specification of their usage is very similar.

No check is done that the called segment actually exists so it is the user's responsibility to ensure this. The called segment will be paged into the correct mapper chip page, but it is the user's responsibility to ensure that the mapper slot is enabled in this page, since neither of these routines will alter the slot selection at all. This is to ensure that they can be fast.

The routine cannot be used to do an inter-segment call into page-3. If this is attempted then the specified address in page-3 will simply be called without any paging, since page-3 must never be altered. Calling into page-0 must be done with some care because of the interrupt and other entry point. Also care must be taken that the stack is not paged out by these calls.

These routines always disable interrupts before passing control to the called routine, but do not disable them on return. It is therefore the responsibility of the called routine to re-enable interrupts as soon as possible. It is also advisable to put an "EI" instruction after the inter-segment call just in case the called routine did not re-enable interrupts.

Parameters cannot be passed in registers IX, IY, AF', BC', DE' or HL' since these are used internally in the routine. These registers will be corrupted by the inter-segment call and may also be corrupted by the called routine. All other registers (AF, BC, DE and HL) will be passed intact to the called routine and returned from it to the caller.

CAL_SEG - Parameters: IYh = segment number to be called
IX = address to call
Results: AF,BC,DE,HL returned from called routine. All others corrupted.

CALLS - Parameters: None
Calling sequence:

```
CALL  CALLS
DB    SEGMENT
DW    ADDRESS
```

Results: AF,BC,DE,HL returned from called routine. All others corrupted.

5.7 DIRECT PAGING ROUTINES

The following routines are provided to allow programs to directly manipulate the current paging state without having to access the hardware. Using these routines ensures compatibility with any changes to the details of the hardware. The routines are very fast and so using them will not compromise the performance of programs.

Routines are provided to directly read or write to any of the four page registers. These routines are equivalent to simple Z-80 "IN" or "OUT" instruction to the page registers. No checking of the validity of the segment number is done so this is the user's responsibility. The "GET" routines return values from internal images of the registers without actually reading the registers themselves. This ensures that if a segment is enabled by, for example, "PUT_P1" then a subsequent "GET_P1" call will return the actual value. Reading the mapper register may produce a different value because the top bits of the segment numbers are generally not recorded.

Although a "PUT_P3" routine is provided, it is in fact a dummy routine and will not alter the page-3 register. This is because the contents of the page-3 register should never be altered. The "GET_P3" routine does behave as expected to allow the user to determine what segment is in page-3.

Another pair of routines ("GET_PH" and "PUT_PH") is provided which are identical in function except that the page is specified by the top two bits of register H. This is useful when register HL contains an address, and these routines do not corrupt register HL. "PUT_PH" will never alter the page-3 register.

PUT_Pn -	Parameters:	n = 0,1,2 or 3 to select page A = segment number
	Results:	None All registers preserved
GET_Pn -	Parameters:	n = 0,1,2 or 3 to select page
	Results:	A = segment number All other registers preserved
PUT_PH -	Parameters:	H = high byte of address A = segment number
	Results:	None All registers preserved
GET_PH -	Parameters:	H = high byte of address
	Results:	A = segment number All other registers preserved

Before using these direct paging routines to alter the paging state, a program should first use the "GET_Pn" routines to determine the initial four segments for when it needs to restore these. No program should assume fixed values for these initial segments since they are likely to change in future versions of the system.

6. ERRORS

All the new MSX-DOS functions (function codes above 40h) return an "error code" in A. This is zero if the operation was successful. If non-zero, then the error code explains the exact nature of the error.

Since MSX-DOS performs an "OR A" instruction immediately before returning from a function call, a "JR NZ" instruction is often used in the transient program immediately after the "CALL 5" instruction to test whether an error occurred. Frequently the destination of this error jump just loads the error code into B and does a "Terminate with Error Code" function. This then passes the error code back to the command interpreter which prints the appropriate message.

A transient program may also itself get the actual message for any error returned by an MSX-DOS function call by using the "Explain Error Code" function. See the Function Specification for details.

The error codes start at 0FFh and descend in value. Values less than 40h are 'user errors' and will never be used by the system and can be used by transient programs to return their own errors. User errors below 20h returned to the command interpreter will not have any message printed.

If the "Explain Error Code" function call (see the Function Specification) is asked to explain an error code for which it does not have a message, then the string returned will be "System error <n>" or "User error <n>" as appropriate, where <n> is the error number.

Below is a list of all currently defined error numbers and their messages and meanings. Also given is the mnemonic, which is often used as a symbol in a source file and is used throughout the MSX-DOS system to refer to a particular error.

6.1 DISK ERRORS

The errors in this group are those which are usually passed to disk error handling routines. By default they will be reported as "Abort, Retry" errors. These errors will not normally be returned from MSX-DOS function calls other than "format disk".

Incompatible disk (.NCOMP, 0FFh)

The disk cannot be accessed in that drive (eg. a double sided disk in a single sided drive).

Write error (.WRERR, 0FEh)

General error occurred during a disk write.

Disk error (.DISK, 0FDh)

General unknown disk error occurred.

Not ready (.NRDY, 0FCh)

Disk drive did not respond, usually means there is no disk in the drive.

Verify error (.VERFY, 0FBh)

With VERIFY enabled, a sector could not be read correctly after being written.

Data error (.DATA, 0FAh)

A disk sector could not be read because the CRC error checking was incorrect, usually indicating a damaged disk.

Sector not found (.RNF, 0F9h)

The required sector could not be found on the disk, usually means a damaged disk.

Write protected disk (.WPROT, 0F8h)

Attempt to write to a disk with the write protect tab on.

Unformatted disk (.UFORM, 0F7h)

The disk has not been formatted, or it is a disk using a different recording technique.

Not a DOS disk (.NDOS, 0F6h)

The disk is formatted for another operating system and cannot be accessed by MSX-DOS.

Wrong disk (.WDISK, 0F5h)

The disk has been changed while MSX-DOS was accessing it. Must replace the correct disk.

Wrong disk for file (.WFILE, 0F4h)

The disk has been changed while there is an open file on it. Must replace the correct disk.

Seek error (.SEEK, 0F3h)

The required track of the disk could not be found.

Bad file allocation table (.IFAT, 0F2h)

The file allocation table on the disk has got corrupted. CHKDSK may be able to recover some of the data on the disk.

(.NOUPB, 0F1h)

This error has no message because it is always trapped internally in MSX-DOS as part of the disk change handling.

Cannot format this drive (.IFORM, 0F0h)

Attempt to format a drive which does not allow formatting. Usually as a result of trying to format the RAM disk.

6.2 MSX-DOS FUNCTION ERRORS

The following errors are those which are normally returned from MSX-DOS function calls. See the Function Specification document for details of errors from particular MSX-DOS functions.

Internal error (.INTER, 0DFh)

Should never occur.

Not enough memory (.NORAM, 0DEh)

MSX-DOS has run out of memory in its 16k kernel data segment. Try reducing the number of sector buffers or removing some environment strings. Also occurs if there are no free segments for creating the RAMdisk.

Invalid MSX-DOS call (.IBDOS, 0DCh)

An MSX-DOS call was made with an illegal function number. Most illegal function calls return no error, but this error may be returned if a "get previous error code" function call is made.

Invalid drive (.IDRV, 0DBh)

A drive number parameter, or a drive letter in a drive/path/file string is one which does not exist in the current system.

Invalid filename (.IFNM, 0DAh)

A filename string is illegal. This is only generated for pure filename strings, not drive/path/file strings.

Invalid pathname (.IPATH, 0D9h)

Can be returned by any function call which is given an ASCIIZ drive/path/file string. Indicates that the syntax of the string is incorrect in some way.

Pathname too long (.PLONG, 0D8h)

Can be returned by any function call which is given an ASCIIZ drive/path/file string. Indicates that the complete path being specified (including current directory if used) is longer than 63 characters.

File not found (.NOFIL, 0D7h)

Can be returned by any function which looks for files on a disk if it does not find one. This error is also returned if a directory was not found, unless the directory was part of a pathname.

Directory not found (.NODIR, 0D6h)

Returned if a directory item in a drive/path/file string could not be found.

Root directory full (.DRFUL, 0D5h)

Returned by "create" or "move" if a new entry is required in the root directory and it is already full. The root directory cannot be extended.

Disk full (.DKFUL, 0D4h)

Usually results from a write operation if there was insufficient room on the disk for the amount of data being written. May also result from trying to create or extend a sub-directory if the disk is completely full.

Duplicate filename (.DUPF, 0D3h)

Results from "rename" or "move" if the destination filename already exists in the destination directory.

Invalid directory move (.DIRE, 0D2h)

Results from an attempt to move a sub-directory into one of its own descendants. This is not allowed as it would create an isolated loop in the directory structure.

Read only file (.FILRO, 0D1h)

Attempt to write to or delete a file which has the "read only" attribute bit set.

Directory not empty (.DIRNE, 0D0h)

Attempt to delete a sub-directory which is empty.

Invalid attributes (.IATTR, 0CFh)

Can result from an attempt to change a fi attributes in an illegal way, or trying to do operation on a file which is only possible on a directory. Also results from illegal use of vo name fileinfo blocks.

Invalid . or .. operation (.DOT, 0CEh)

Attempt to do an illegal operation on the ". "." entries in a sub-directory, such as rename or them.

System file exists (.SYSX, 0CDh)

Attempt to create a file or sub-directory of same name as an existing system file. System file not automatically deleted.

Directory exists (.DIRX, 0CCh)

Attempt to create a file or sub-directory of same name as an existing sub-directory. directories are not automatically deleted.

File exists (.FILEX, 0CBh)

Attempt to create a sub-directory of the same as an existing file. Files are not automati deleted when creating sub-directories.

File already in use (.FOPEN, 0CAh)

Attempt to delete, rename, move, or chang attributes or date and time of a file which has handle already open to it, other than by using th handle itself.

Copyright (1986) ASCII Corp. 86/11/26 (version 2.00) p

Cannot transfer above 64K (.OV64K, 0C9h)

Disk transfer area would have extended above 0FFFFh.

File allocation error (.FILE, 0C8h)

The cluster chain for a file was corrupt. Use CHKDSK to recover as much of the file as possible.

End of file (.EOF, 0C7h)

Attempt to read from a file when the file pointer is already at or beyond the end of file.

File access violation (.ACCV, 0C6h)

Attempt to read or write to a file handle which was opened with the appropriate access bit set. Some of the standard file handles are opened in read only or write only mode.

Invalid process id (.IPROC, 0C5h)

Process id number passed to "join" is invalid.

No spare file handles (.NHAND, 0C4h)

Attempt to open or create a file handle when all file handles are already in use. There are 128 file handles available.

Invalid file handle (.IHAND, 0C3h)

The specified file handle is greater than the maximum allowed file handle number of 127.

File handle not open (.NOPEN, 0C2h)

The specified file handle is not currently open.

Invalid device operation (.IDEV, 0C1h)

Attempt to use a device file handle or fileinfo block for an invalid operation such as searching in it or moving it.

Invalid environment string (.IENV, 0C0h)

Environment item name string contains an invalid character.

Environment string too long (.ELONG, 0BFh)

Environment item name or value string is either longer than the maximum allowed length of 255, or is too long for the user's buffer.

Invalid date (.IDATE, 0BEh)

Date parameters passed to "set date" are invalid.

Invalid time (.ITIME, 0BDh)

Time parameters passed to "set time" are invalid.

RAM disk (drive H:) already exists (.RAMDX, 0BCh)

Returned from the "ramdisk" function if trying to create a RAM disk when one already exists.

RAM disk does not exist (.NRAMD, 0BBh)

Attempt to delete the RAM disk when it does not currently exist. A function which tries to access a non-existent RAM disk will get a .IDRV error.

File handle has been deleted (.HDEAD, 0BAh)

The file associate with a file handle has been deleted so the file handle can no longer be used.

(.EOL, 0B9h)

Internal error should never occur.

Invalid sub-function number (.ISBFN, 0B8h)

The sub-function number passed to the IOCTL function (function 4Bh) was invalid.

6.3 PROGRAM TERMINATION ERRORS

The following errors are those which may be generated internally in the system and passed to "abort" routines. They will not normally be returned from function calls. Note that an abort routine can also be passed any error which a transient program passes to the "terminate with error code" function call.

Ctrl-STOP pressed (.STOP, 09Fh)

The Ctrl-STOP key is tested in many places in the system including all character I/O and on all calls to disk drivers.

Ctrl-C pressed (.CTRLC, 09Eh)

Ctrl-C is only tested for on those character functions which specify status checks.

Disk operation aborted (.ABORT, 09Dh)

This error occurs when any disk error is aborted by the user or automatically by the system. The original disk error code will be passed to the abort routine in B as the secondary error code.

Error on standard output (.OUTERR, 09Ch)

Returned if any error occurred on a standard output channel while it was being accessed through the character functions (functions 01h...0Bh). The original error code is passed to the abort routine in register B as the secondary error code. This error will normally only occur if a program has altered the standard file handles.

Error on standard input (.INERR, 09Bh)

Returned if any error occurred on a standard input channel while it was being accessed through the character functions (functions 01h...0Bh). The original error code is passed to the abort routine in register B as the secondary error code. The most likely error is end of file (.EOF). This error will normally only occur if a program has altered the standard file handles.

6.4 COMMAND ERRORS

The following errors will not be returned from an MSX-DOS function call, but are used by the command interpreter. They are included here because a transient program may find it useful to return some of them. The "Command Specification" document gives more details of what these errors means from the command interpreter.

Wrong version of COMMAND2.COM (.BADCOM, 08Fh)

COMMAND2.COM loaded it's transient part from disk but it's checksum was not what was expected.

Unrecognized command (.BADCM, 08Eh)

A given command was not an internal command and a .COM or .BAT file was not found with the same name.

Command too long (.BUFUL, 08Dh)

The command in a batch file exceeded 127 characters in length.

(.OKCMD, 08Ch)

An internal error used after executing a command passed to COMMAND2.COM on the command line. (There is no message for this error code.)

Invalid parameter (.IPARM, 08Bh)

The parameter to a command was invalid in some way eg. a number out of range.

Too many parameters (.INP, 08Ah)

After parsing all the parameters required for a command, there were still more non-separator characters on the command line.

Missing parameter (.NOPAR, 089h)

Where a parameter was expected the end of line was found.

Invalid option (.IOPT, 088h)

The letter given after a / on the command line was invalid for that command.

Invalid number (.BADNO, 087h)

A number contained non-digit characters.

File for HELP not found (.NOHELP, 086h)

The help file was not found or the parameter was not a valid HELP parameter.

Wrong version of MSX-DOS (.BADVER, 085h)

This error is never used by the command interpreter, it has its own internal message for this error. However it is provided for transient programs which may find it useful to return this error.

Cannot concatenate destination file (.NOCAT, 084h)

The destination file in CONCAT is matched by the source specification.

Cannot create destination file (.BADEST, 083h)

In COPY, creating the destination file would overwrite one of the source files (or another file that is already in use).

File cannot be copied onto itself (.COPY, 082h)

In COPY, the destination file if created would overwrite the source file.

Cannot overwrite previous destination file (.OVDEST, 081h)

In COPY, an ambiguous source was specified with a non-ambiguous, non-directory, non-device destination.

* * * * *